Deep Learning Specialization
Neural Networks and Deep Learning
Learning Notes

Du Ang
du2ang233@gmail.com

October 2, 2017

Contents

1 Welcome

AI is the new Electricity. — Andrew Ng

Electricity had once transformed countless industries: transportation, manufacturing, healthcare, communications, and more.

AI will now bring about an equally big transformation.

Courses in this sequence (Specialization):

1. Neural Networks and Deep Learning
2. Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization
3. Structuring Machine Learning Projects
4. Convolutional Neural Networks
5. Sequence Models

2 Introduction to Deep Learning

2.1 What is a neural network?

It is powerful learning algorithm inspired by how the brain works.

2.1.1 Single neural network

Given data about the size of houses on the real estate market and you want to fit a function that will predict their price. It is a linear regression problem because the price as a function of size is a continuous output.

We know the prices can never be negative so we are creating a function called Rectified Linear Unit (ReLU) which starts at zero.

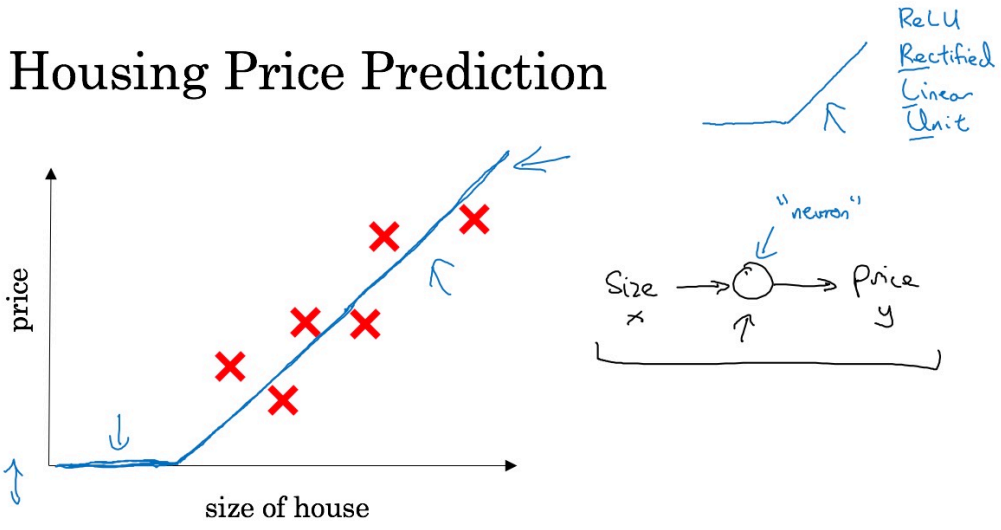


Figure 1: The "Housing Price Prediction" problem. The input is the size of the house (x); The output is the price (y); The "neuron" implements the function ReLU.

2.1.2 Multiple neural network

The price of a house can be affected by other features such as size, number of bedrooms, zip code and wealth. The role of the neural network is to predicted the price and it will automatically generate the hidden units. We only need to give the inputs x and the output y.

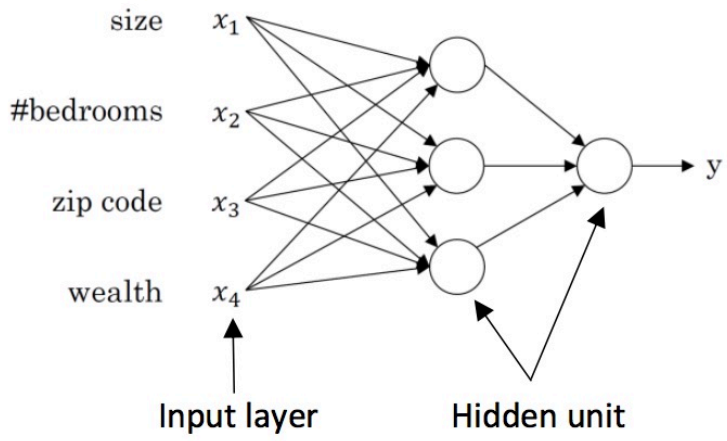


Figure 2: Multiple neural network

2.2 Supervised Learning

In supervised learning, we are given a data set and already know about what our correct output should look like, having the idea that there is a relationship between the input and the output.

Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, we are trying to predict results with a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

Here are some example of supervised learning:

Table 1: Supervised Learning Applications

Input(x)	**Output**(y)	**Application**	**NN Types**
Home features	Price	Real Estate	Standard NN
Ad, user info	Click on ad? (0/1)	Online Advertising	Standard NN
Image	Object (1, ..., 1000)	Photo tagging	CNN
Audio	Text transcript	Speech Recognition	RNN
English	Chinese	Machine translation	RNN
Image, Radar info	Position of other cars	Autonomous driving	Custom/Hybrid

There are different types of neural network, for example, Convolutional Neural Network (CNN) used often for image application and Recurrent Neural Network (RNN) used for one-dimensional sequence data such as translating English to Chinese or a temporal component such as text transcript. As for the autonomous driving, it is a hybrid neural network architecture.

2.2.1 Structured vs unstructed data

Structed data refers to things that has a defined meaning such as price, age whereas unstructed data refers to thing like pixel, raw audio, text.

2.3 Why is Deep Learning taking off?

Deep learning is taking off due to a large amount of data available through the digitization of the society, faster computation and innovation in the development of neural network algorithm.

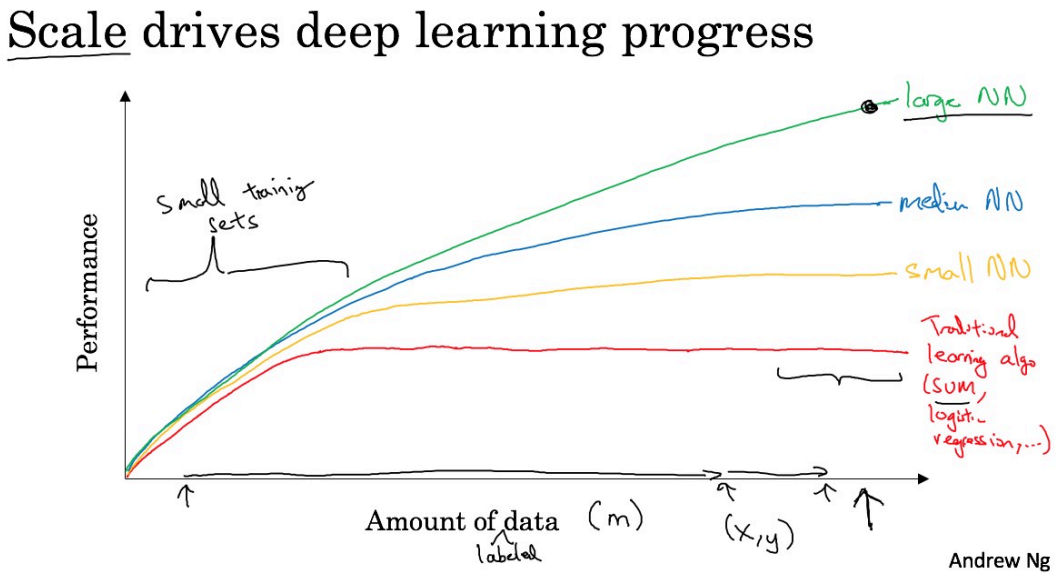
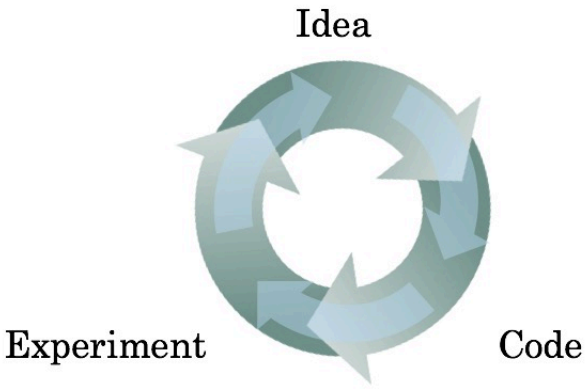


Figure 3: Scale drives deep learning progress

Two things have to be considered to get to the high level of performance:

1. Being able to train a big enough neural network
2. Huge amount of labelled data

The process of training a neural network is iterative:



It could take a good amount of time to train a neural network, which affects your productivity. Faster computation helps to iterate and improve new algorithm.

3 Neural Network Basics

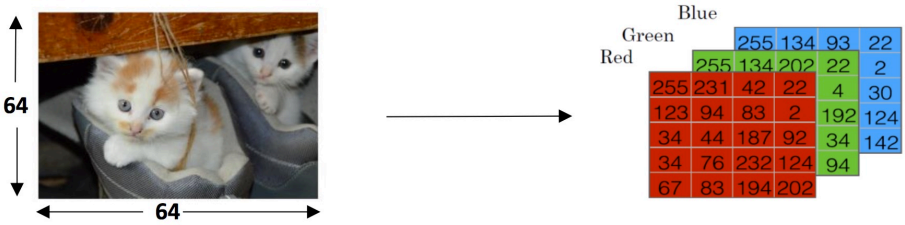
3.1 Logistic Regression as a Neural Network

3.1.1 Binary Classification

In a binary classification problem, the result is a discrete value output. For example:

- account hacked (1) or compromised (0)
- a tumor malign (1) or benign (0)

Example: Cat vs Non-Cat The goal is to train a classifier that the input is an image represented by a feature vector, x, and predicts whether the corresponding label y is 1 or 0. In this case, whether this is a cat image (1) or a non-cat image (0).



An image is store in the computer in three separate matrices corresponding to the Red, Green, and Blue color channels of the image. The three matrices have the same size as the image, for example, the resolution of the cat image is 64 pixels $\times$ 64 pixels, the three matrices (RGB) are 64 $\times$ 64 each.

The value in a cell represents the pixel intensity which will be used to create a feature vector of n-dimension. In pattern recognition and machine learning, a feature vector represents an object, in this case, a cat or no cat.

To create a feature vector, $\boldsymbol{x}$, the pixel intensity values will be "unroll" or "reshape" for each color. The dimension of the input feature vector $\boldsymbol{x}$, is $n_x = 64 \times 64 \times 3 = 12288$.

$$x = \begin{bmatrix} 255 \\ 231 \\ 42 \\ \vdots \\ 255 \\ 134 \\ 202 \\ \vdots \\ 255 \\ 134 \\ 93 \\ \vdots \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \\ \\ \text{red} \\ \\ \\ \\ \text{green} \\ \\ \\ \\ \text{blue} \\ \\ \end{matrix}$$

3.1.2 Notation

single example: $(\boldsymbol{x}, y)$, $\boldsymbol{x} \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$
m training examples: $\{(\boldsymbol{x}^{(1)}, y^{(2)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$
$m = m_{train}$ $\quad$ m_{test} = the number of test examples

$$\boldsymbol{X} = \begin{bmatrix} | & | & & | \\ \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \cdots & \boldsymbol{x}^{(m)} \\ | & | & & | \end{bmatrix} \qquad \boldsymbol{X} \in \mathbb{R}^{n_x \times m}$$

$$\boldsymbol{Y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} \qquad \boldsymbol{Y} \in \mathbb{R}^{1 \times m}$$

In Python/NumPy, `X.shape = (n_x, m), Y.shape = (1, m).`

3.1.3 Logistic Regression

Logistic regression is a learning algorithm used in a supervised learning problem when the output y are all either zero or one. The goal of logistic regression is to minimize the error between its predictions and training data.

Example: Cat vs Non-Cat Given an image represented by a feature vector $\boldsymbol{x}$, the algorithm will evaluate the probabilty of a cat being in that image.

Given $\boldsymbol{x}$, want $\hat{y} = P(y = 1|\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^{n_x}$, $0 \leq \hat{y} \leq 1$

Parameters: $\boldsymbol{w} \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output: ~~$\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b$~~ $\hat{y} = \sigma(\boldsymbol{w}^T\boldsymbol{x} + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$.

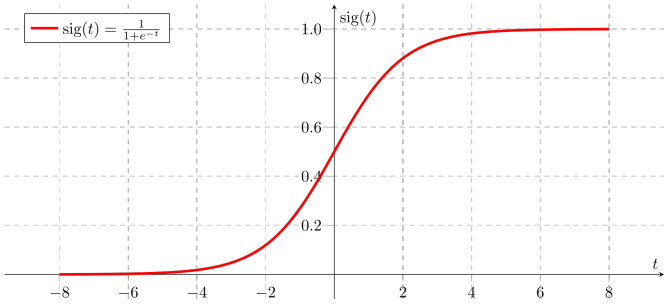


Figure 4: Sigmoid

$(\boldsymbol{w}^T\boldsymbol{x} + b)$ is a linear function $(a\boldsymbol{x} + b)$, but since we are looking for a probability constraint between [0, 1], the sigmoid function is used. The function is bounded between [0, 1] as shown in the graph 4.

Some observations from the sigmoid function graph:

- If z is a large positive number, then $\sigma(z) = 1$
- If z is small or large negative number, then $\sigma(z) = 0$
- If $z = 0$, then $\sigma(z) = 0.5$

Logistic Regression cost function $\hat{y}^{(i)} = \sigma(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$

Given $\{(\boldsymbol{x}^{(1)}, y^{(2)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$.

Loss (error) function: (for single example)

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

If $y = 1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y} \leftarrow$ want $\log \hat{y}$ large, want $\hat{y}$ large.

If $y = 0$: $\mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y}) \leftarrow$ want $\log(1 - \hat{y})$ large, want $\hat{y}$ small.

The loss function measures the discrepancy between the prediction ($\hat{y^{(i)}}$) and the desired output ($y^{(i)}$). In other words, the loss function computes the error for a single training example.

Note: Why can't use $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$? Because $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$ is convex, however, $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ is non-convex.

Cost function: (for entire training set)

$$J(\boldsymbol{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[(y^{(i)}\log(\hat{y}^{(i)})) + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$$

The cost function is the average of the loss function of the entire training set. We are going to find the parameters $\boldsymbol{w}$ and b that minimize the overall cost function.

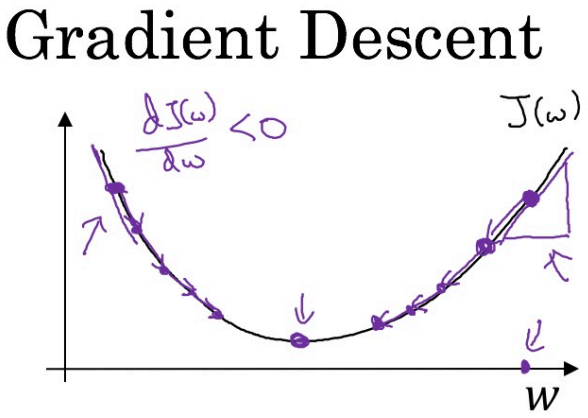


Figure 5: Gradient descent. (Ignore b here)

repeat

$$\boldsymbol{w} := \boldsymbol{w} - \alpha\frac{\partial J(\boldsymbol{w}, b)}{\partial \boldsymbol{w}}$$
$$b := b - \alpha\frac{\partial J(\boldsymbol{w}, b)}{\partial b}$$

until converge;

Gradient Descent Note: We often use "d$\boldsymbol{w}$" to denote $\frac{\partial J(\boldsymbol{w}, b)}{\partial \boldsymbol{w}}$, use "d$b$" to denote $\frac{\partial J(\boldsymbol{w}, b)}{\partial b}$. In other words, we use "dvar" to denote $\frac{\mathrm{d}FinalOutputVar}{\mathrm{d}var}$ or $\frac{\partial FinalOutputVar}{\partial var}$.

Gradient on single example:

$$z = \boldsymbol{w}^T\boldsymbol{x} + b$$

$$\hat{y} = a = \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\mathcal{L}(a, y) = -(y\log(a) + (1-y)\log(1-a))$$

According to the computation graph 6, go backwards to compute the derivatives:

$$\mathrm{d}a = \frac{\mathrm{d}\mathcal{L}(a, y)}{\mathrm{d}a} = -\frac{y}{a} + \frac{1-y}{1-a}$$

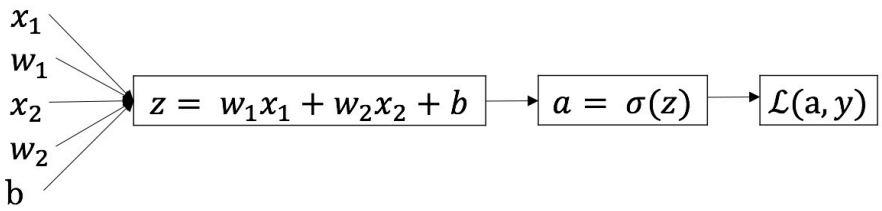


Figure 6: The computation graph of logistic regression gradient descent

$$\mathrm{d}z = \frac{\mathrm{d}\mathcal{L}(a,y)}{\mathrm{d}z} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}a} \cdot \frac{\mathrm{d}a}{\mathrm{d}z} = (-\frac{y}{a} + \frac{1-y}{1-a}) \cdot a(1-a) = a - y$$

$$\mathrm{d}w_1 = \frac{\partial \mathcal{L}}{\partial w_1} = x_1 \mathrm{d}z = x_1(a-y)$$

$$\mathrm{d}w_2 = \frac{\partial \mathcal{L}}{\partial w_2} = x_2 \mathrm{d}z = x_2(a-y)$$

$$\mathrm{d}b = \frac{\partial \mathcal{L}}{\partial b} = \mathrm{d}z = a - y$$

Gradient on m examples:

$$J(\boldsymbol{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

$$a^{(i)} = \hat{y^{(i)}} = \sigma(z^{(i)}) = \sigma(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b)$$

$$\frac{\partial J(\boldsymbol{w}, b)}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}(a^{(i)}, y^{(i)})}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} \mathrm{d}w_1^{(i)}$$

The Algorithm 1 is the algorithm of logistic regression gradient descent on m examples. It uses two for-loops, so it's less efficient than use vertorization.

// Initialization
$J = 0$;
for $j = 1$ **to** n **do**
  $\mathrm{d}w_j = 0$
end
$\mathrm{d}b = 0$;
// Compute cost and derivatives
for $i = 0$ **to** m **do**
  $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)} + b$;
  $a^{(i)} = \sigma(z^{(i)})$;
  $J = J - [y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$;
  $\mathrm{d}z^{(i)} = a^{(i)} - y^{(i)}$;
  for $j = 1$ **to** n **do**
    $\mathrm{d}w_j = \mathrm{d}w_j + x_j^{(i)} \mathrm{d}z^{(i)}$;
  end
  $\mathrm{d}b = \mathrm{d}b + \mathrm{d}z^{(i)}$
end
// Get the average
$J = J/m$;
for $j = 1$ **to** n **do**
  $\mathrm{d}w_j = \mathrm{d}w_j/m$
end
$\mathrm{d}b = \mathrm{d}b/m$;
// Gradient descent
for $j = 1$ **to** n **do**
  $\mathrm{d}w_j = \mathrm{d}w_j - \alpha \mathrm{d}w_j$
end
$b = b - \alpha \mathrm{d}b$

Algorithm 1: Logistic regression gradient descent on m examples

3.2 Python and Vectorization

3.2.1 Vectorization

What is vectorization?

$$z = \boldsymbol{w}^T \boldsymbol{x} + b \qquad \boldsymbol{w} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}, \qquad \boldsymbol{x} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}, \qquad \boldsymbol{w} \in \mathbb{R}^{n_x}, \quad \boldsymbol{x} \in \mathbb{R}^{n_x}$$

Non vectorized:

```
z = 0
for i in range(n_x):
    z += w[i] * x[i]
z += b
```

Vectorized:

```
z = np.dot(w, x) + b
```

$$\begin{cases} \text{CPU} \\ \text{GPU} \end{cases} \qquad \text{SIMD - Single Instruction Multiple Data}$$

3.2.2 More Vectorization Examples

Neural network programming guideline Whenever possible, avoid explicit for-loops.

For example, compute $\boldsymbol{u} = \boldsymbol{A}\boldsymbol{v}$. $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{v} \in \mathbb{R}^n$.

Non-vectorized: $u_i = \sum_j \boldsymbol{A}_{ij} v_j$

```
u = np.zeros((n, 1))
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        u[i] += A[i][j] * v[j]
```

Vectorized:

```
u = np.dot(A, v)
```

Vectors and matrix valued functions Say you need to apply the exponential operation on every element of a matrix/vector.

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \longrightarrow \boldsymbol{u} = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

Non-vectorized:

```
u = np.zeors((n, 1))
for i in range(n):
    u[i] = math.exp(v[i])
```

Vectorized:

```
u = np.exp(v)
```

$J = 0$
// $\mathrm{d}w_1 = 0, \mathrm{d}w_2 = 0, \ldots, \mathrm{d}w_{n_x}$
$\mathrm{d}\boldsymbol{w} = \text{np.zeros}((n_x, 1))$
$\mathrm{d}b = 0$
for $i = 1$ **to** m **do**
  $z^{(i)} = \boldsymbol{w}^T x^{(i)} + b$
  $a^{(i)} = \sigma(z^{(i)})$
  $J = J - [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$
  $\mathrm{d}z^{(i)} = a^{(i)} - y^{(i)}$
  // $\mathrm{d}w_1 = \mathrm{d}w_1 + x_1^{(i)} \mathrm{d}z^{(i)}$
  // $\mathrm{d}w_2 = \mathrm{d}w_2 + x_2^{(i)} \mathrm{d}z^{(i)}$
  // …
  // $\mathrm{d}w_{n_x} = \mathrm{d}w_{n_x} + x_{n_x}^{(i)} \mathrm{d}z^{(i)}$
  $\mathrm{d}\boldsymbol{w} = \mathrm{d}\boldsymbol{w} + x^{(i)} \mathrm{d}z^{(i)}$
  $\mathrm{d}b = \mathrm{d}b + \mathrm{d}z^{(i)}$
end
$J = J/m$
// $\mathrm{d}w_1 = \mathrm{d}w_1/m$
// $\mathrm{d}w_2 = \mathrm{d}w_2/m$
// …
// $\mathrm{d}w_{n_x} = \mathrm{d}w_{n_x}/m$
$\mathrm{d}\boldsymbol{w} = \mathrm{d}\boldsymbol{w}/m$
$\mathrm{d}b = \mathrm{d}b/m$

Algorithm 2: Replace the inner for-loop by vectorization in logistic regression

Logistic regression derivatives

3.2.3 Vectorizing Logistic Regression

Inference process:

$$z^{(1)} = \boldsymbol{w}^T \boldsymbol{x^{(1)}} + b \qquad z^{(2)} = \boldsymbol{w}^T \boldsymbol{x^{(2)}} + b \qquad \cdots \qquad z^{(m)} = \boldsymbol{w}^T \boldsymbol{x^{(m)}} + b$$

$$a^{(1)} = \sigma(z^{(1)}) \qquad a^{(2)} = \sigma(z^{(2)}) \qquad \cdots \qquad a^{(m)} = \sigma(z^{(m)})$$

$$\boldsymbol{X} = \begin{bmatrix} | & | & & | \\ \boldsymbol{x^{(1)}} & \boldsymbol{x^{(2)}} & \cdots & \boldsymbol{x^{(m)}} \\ | & | & & | \end{bmatrix} \qquad \boldsymbol{X} \in \mathbb{R}^{n_x \times m}$$

$$\boldsymbol{Z} = \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} = \boldsymbol{w}^T \boldsymbol{X} + \underbrace{\begin{bmatrix} b & b & \cdots & b \end{bmatrix}}_{1 \times m}$$

$$\boldsymbol{A} = \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix} = \sigma(\boldsymbol{Z})$$

$$\mathrm{d}z^{(1)} = a^{(1)} - y^{(1)} \qquad \mathrm{d}z^{(2)} = a^{(2)} - y^{(2)} \qquad \cdots \qquad \mathrm{d}z^{(m)} = a^{(m)} - y^{(m)}$$

$$\mathrm{d}\boldsymbol{Z} = \begin{bmatrix} \mathrm{d}z^{(1)} & \mathrm{d}z^{(2)} & \cdots & \mathrm{d}z^{(m)} \end{bmatrix}$$

$$\boldsymbol{Y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

$$\mathrm{d}\boldsymbol{Z} = \boldsymbol{A} - \boldsymbol{Y} = \begin{bmatrix} a^{(1)} - y^{(1)} & a^{(2)} - y^{(2)} & \cdots & a^{(m)} - y^{(m)} \end{bmatrix}$$

$$\mathrm{d}b = \frac{1}{m} \sum_{i=1}^{m} \mathrm{d}z^{(i)} \longrightarrow \mathrm{d}b = \frac{1}{m} \mathrm{np.sum}(\mathrm{d}\boldsymbol{Z})$$

$$\mathrm{d}\boldsymbol{w} = \frac{1}{m} \begin{bmatrix} | & | & & | \\ \boldsymbol{x^{(1)}} & \boldsymbol{x^{(2)}} & \cdots & \boldsymbol{x^{(m)}} \\ | & | & & | \end{bmatrix} \begin{bmatrix} \mathrm{d}z^{(1)} \\ \mathrm{d}z^{(2)} \\ \vdots \\ \mathrm{d}z^{(m)} \end{bmatrix} = \frac{1}{m} \underbrace{\left[\boldsymbol{x}^{(1)} \mathrm{d}z^{(1)} + \boldsymbol{x}^{(2)} \mathrm{d}z^{(2)} + \cdots + \boldsymbol{x}^{(m)} \mathrm{d}z^{(m)} \right]}_{n \times 1} \longrightarrow \mathrm{d}\boldsymbol{w} = \frac{1}{m} \boldsymbol{X} \mathrm{d}\boldsymbol{Z}^T$$

Vectorized (in a single iteration):

$$\boldsymbol{Z} = \boldsymbol{w}^T \boldsymbol{X} + b \longrightarrow \mathrm{np.dot(w.T,\ X)\ +\ b}$$

$$\boldsymbol{A} = \sigma(\boldsymbol{Z})$$

$$\mathrm{d}\boldsymbol{Z} = \boldsymbol{A} - \boldsymbol{Y}$$

$$\mathrm{d}\boldsymbol{w} = \frac{1}{m} \boldsymbol{X} (\mathrm{d}\boldsymbol{Z}^T)$$

$$\mathrm{d}b = \frac{1}{m} \mathrm{np.sum}(\mathrm{d}\boldsymbol{Z})$$

Then update $\boldsymbol{w}$ and b for each iteration:

$$\boldsymbol{w} = \boldsymbol{w} - \alpha \mathrm{d}\boldsymbol{w}$$

$$b = b - \alpha \mathrm{d}b$$

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

Figure 7: Calories from Carbs, Proteins, Fats in 100g of different foods

3.2.4 Broadcasting in Python

According to Figure 7, denote the matrix in the figure as $\boldsymbol{A}$, calculate the percentages of calories from Carbs, Proteins, Fats for each of four foods. Try do that without explicit for-loop.

```
cal = A.sum(axis=0) # sum vertically, pass axis=1 to sum horizontally
# broadcasting, A.shape = (3, 4), cal.shape = (1, 4)
percentage = 100 * A / (cal.reshape(1, 4))
```

Broadcasting examples

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 100 & 100 \\ 200 & 200 & 200 \end{bmatrix}$$

(Note: In MATLAB/Octave, the `bsxfun` seems to have the same feature, it can apply element-wise operation to two arrays with implicit expansion enabled.)

General principle of broadcasting

$$\underbrace{\boldsymbol{Matrix}}_{(m,n)} \quad \begin{matrix} + \\ - \\ * \\ / \end{matrix} \quad \underbrace{\boldsymbol{Vector}}_{(1,n)or(m,1)} \quad \longrightarrow \quad \underbrace{\boldsymbol{Matrix}}_{(m,n)}$$

3.2.5 A Note on Python/NumPy Vectors

For broadcasting:

$$\begin{cases} \text{Strenght:} & \text{expressivity, flexibility} \\ \text{Weakness:} & \text{may introduce subtle bugs} \end{cases}$$

```
>>> import numpy as np
>>> a = np.random.randn(5)
>>> a   # rank 1 array, do not use in this course
array([-0.40700705, -1.27728182,  0.10255894,  0.06871343, -1.01614418])
>>> a.shape
(5,)
>>> a = a.reshape((5, 1))        # column vector
```

```
>>> a
array([[-0.40700705],
       [-1.27728182],
       [ 0.10255894],
       [ 0.06871343],
       [-1.01614418]])
>>> a = np.random.randn(5, 1)   # column vector
>>> a
array([[ 1.32238445],
       [-2.09932992],
       [ 0.29040232],
       [-1.31295795],
       [-1.05893313]])
>>> a.shape
(5, 1)
>>> a = np.random.randn(1, 5)   # row vector
>>> a
array([[-0.7020727 , -0.54867968,  1.38088584,  0.69785511, -0.2510519 ]])
>>> a.shape
(1, 5)
>>> assert(a.shape == (1, 5))
>>> assert(a.shape == (5, 1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

3.2.6 Explanation of Logistic Regression Cost Function (Optional)

Logistic regression cost function:

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b), \qquad \text{where } \sigma(z) = \frac{1}{1+e^{-z}}.$$

$$\left.\begin{array}{ll} \text{If } y=1: & P(y|x) = \hat{y} \\ \text{If } y=0: & P(y|x) = 1-\hat{y} \end{array}\right\} P(y|x) = \hat{y}^y (1-\hat{y})^{(1-y)}$$

$$\log P(y|x) = y \log \hat{y} + (1-y) \log(1-\hat{y}) = -\mathcal{L}(\hat{y}, y)$$

Cost on m examples: (Maximum Likelihood Estimation)

$$\log P(\text{labels in training set}) = \log \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) = \underbrace{\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)})}_{-\mathcal{L}(\hat{y}^{(i)}, y^{(i)})} = -\sum_{(i=1)}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$\underset{\text{minimize}}{J(\boldsymbol{w}, b)} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

4 Shallow Neural Network

4.1 Neural Network Overview

Figure 8 shows an example of a shallow neural network.

Notation: Use superscript $[i]$ to denote the i-th layer; use the superscript (i) to denote the i-th example in dataset.

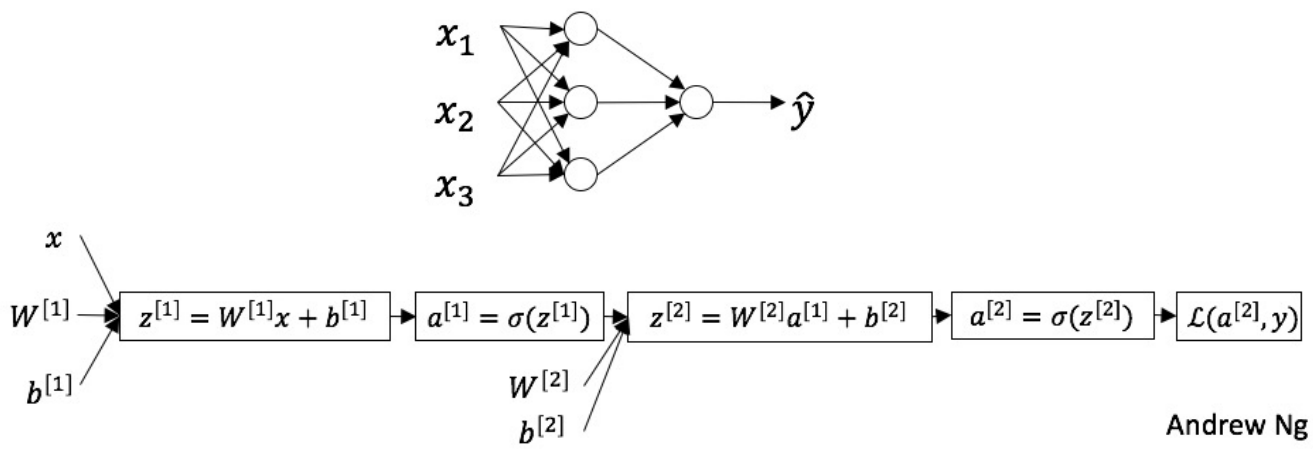


Figure 8: The example of a shallow neural network

4.2 Neural Network Representation

As is shown in Figure 9, $\boldsymbol{a}^{[i]}$ denotes the activation of the i-th layer.

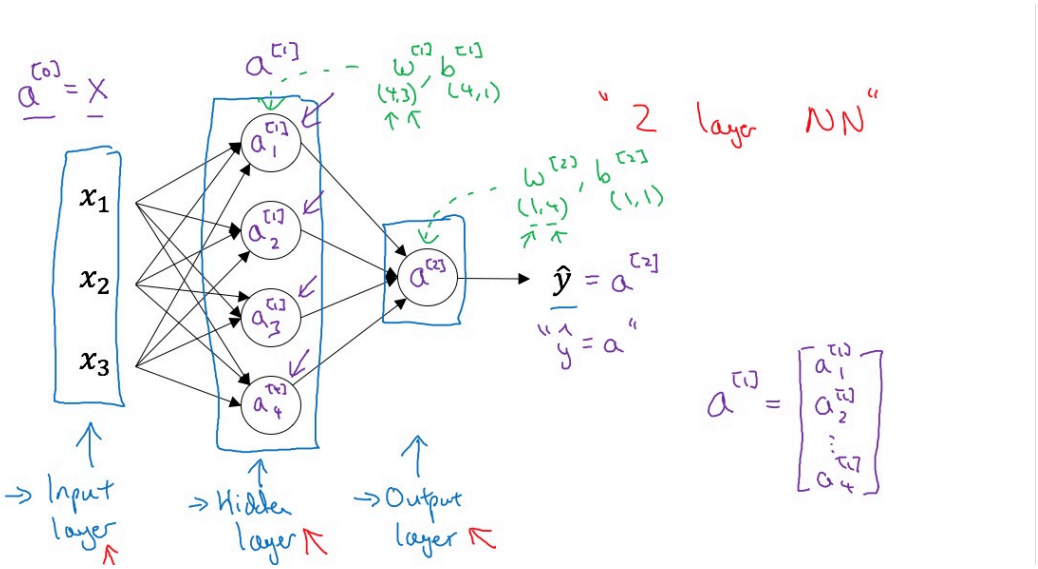


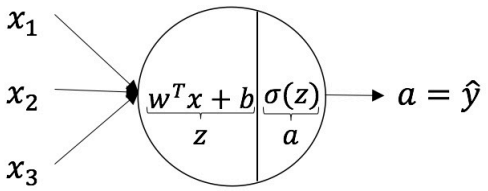
Figure 9: The representation of a neural network

4.3 Computing a Neural Network's Output

As the Figure 10 shows, the compuation of a single neuron includes two parts: the first part, $\boldsymbol{z} = \boldsymbol{w}^T\boldsymbol{x} + b$; the second part, $\boldsymbol{a} = \sigma(\boldsymbol{z})$.

The Figure 11 shows the compuation of the neural network's first layer. Cause it uses a for-loop, it is less efficient. According to it, we can generize a vectorized version:

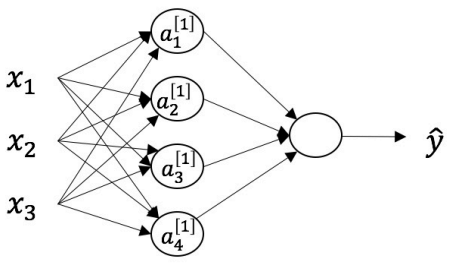
$$\boldsymbol{z}^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \qquad \boldsymbol{W}^{[1]} = \begin{bmatrix} \text{---} \; \boldsymbol{w}_1^{[1]T} \; \text{---} \\ \text{---} \; \boldsymbol{w}_2^{[1]T} \; \text{---} \\ \text{---} \; \boldsymbol{w}_3^{[1]T} \; \text{---} \\ \text{---} \; \boldsymbol{w}_4^{[1]T} \; \text{---} \end{bmatrix} \qquad \boldsymbol{x} = \boldsymbol{a}^{[0]} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$



$$z = w^T x + b$$

$$a = \sigma(z)$$

Figure 10: The computation of a single neuron



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

Figure 11: The computation of the first layer of a neural network

Given input $\boldsymbol{x}(\boldsymbol{a}^{[0]})$:

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{a}^{[0]} + \boldsymbol{b}^{[1]} \qquad \boldsymbol{a}^{[1]} = \sigma(\boldsymbol{z}^{[1]}) \qquad \boldsymbol{z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1]} + \boldsymbol{b}^{[2]} \qquad \boldsymbol{a}^{[2]} = \sigma(\boldsymbol{z}^{[2]})$$

4.4 Vectorizing across Multiple Examples

For single example:

$$\boldsymbol{x} \longrightarrow \boldsymbol{a}^{[2]} = \hat{\boldsymbol{y}}$$

For m examples:

$$\boldsymbol{x}^{(1)} \longrightarrow \boldsymbol{a}^{[2](1)} = \hat{\boldsymbol{y}}^{(1)} \qquad \boldsymbol{x}^{(2)} \longrightarrow \boldsymbol{a}^{[2](2)} = \hat{\boldsymbol{y}}^{(2)} \qquad \ldots \qquad \boldsymbol{x}^{(m)} \longrightarrow \boldsymbol{a}^{[2](m)} = \hat{\boldsymbol{y}}^{(m)}$$

Non-vectorized:

for $i = 1$ **to** m **do**
  $\boldsymbol{z}^{[1](i)} = \boldsymbol{W}^{[1]}\boldsymbol{x}^{(i)} + \boldsymbol{b}^{[1]}$
  $\boldsymbol{a}^{[1](i)} = \sigma(\boldsymbol{z}^{[1](i)})$
  $\boldsymbol{z}^{[2](1)} = \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1](i)} + \boldsymbol{b}^{[2]}$
  $\boldsymbol{a}^{[2](i)} = \sigma(\boldsymbol{z}^{[2](i)})$
end

$$\text{Let} \quad \boldsymbol{X} = \underbrace{\begin{bmatrix} | & | & & | \\ \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \cdots & \boldsymbol{x}^{(m)} \\ | & | & & | \end{bmatrix}}_{(n_x, m)}$$

$$\text{then} \quad \boldsymbol{Z}^{[1]} = \underbrace{\begin{bmatrix} | & | & & | \\ \boldsymbol{z}^{[1](1)} & \boldsymbol{z}^{[1](2)} & \cdots & \boldsymbol{z}^{[1](m)} \\ | & | & & | \end{bmatrix}}_{\text{(hidden units,training examples)}} \qquad \boldsymbol{A}^{[1]} = \underbrace{\begin{bmatrix} | & | & & | \\ \boldsymbol{a}^{[1](1)} & \boldsymbol{a}^{[1](2)} & \cdots & \boldsymbol{a}^{[1](m)} \\ | & | & & | \end{bmatrix}}_{\text{(hidden units,training examples)}}$$

Vectorized:

$$\boldsymbol{Z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{A}^{[0]} + \boldsymbol{b}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{X} + \boldsymbol{b}^{[1]}$$

$$\boldsymbol{A}^{[1]} = \sigma(\boldsymbol{Z}^{[1]})$$

$$\boldsymbol{Z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{A}^{[1]} + \boldsymbol{b}^{[2]}$$

$$\boldsymbol{A}^{[2]} = \sigma(\boldsymbol{Z}^{[2]})$$

4.5 Explanation for Vectorized Implementation

$$\boldsymbol{z}^{[1](1)} = \boldsymbol{W}^{[1]}\boldsymbol{x}^{(1)} + \boldsymbol{b}^{[1]}, \quad \boldsymbol{z}^{[1](2)} = \boldsymbol{W}^{[1]}\boldsymbol{x}^{(2)} + \boldsymbol{b}^{[1]}, \quad \ldots, \quad \boldsymbol{z}^{[1](m)} = \boldsymbol{W}^{[1]}\boldsymbol{x}^{(m)} + \boldsymbol{b}^{[1]}$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} \text{---} & \boldsymbol{w}_1^{[1]T} & \text{---} \\ \text{---} & \boldsymbol{w}_2^{[1]T} & \text{---} \\ \text{---} & \boldsymbol{w}_3^{[1]T} & \text{---} \\ \text{---} & \boldsymbol{w}_4^{[1]T} & \text{---} \end{bmatrix}$$

$$\boldsymbol{W}^{[1]}\boldsymbol{x}^{(1)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix} \qquad \boldsymbol{W}^{[1]}\boldsymbol{x}^{(2)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix} \qquad \cdots \qquad \boldsymbol{W}^{[1]}\boldsymbol{x}^{(m)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

$$\boldsymbol{W}^{[1]}\begin{bmatrix} | & | & & | \\ \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \cdots & \boldsymbol{x}^{(m)} \\ | & | & & | \end{bmatrix} + \boldsymbol{b}^{[1]} = \begin{bmatrix} \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \end{bmatrix} + \boldsymbol{b}^{[1]} = \begin{bmatrix} | & | & & | \\ \boldsymbol{z}^{[1](1)} & \boldsymbol{z}^{[1](2)} & \cdots & \boldsymbol{z}^{[1](m)} \\ | & | & & | \end{bmatrix} = \boldsymbol{Z}^{[1]}$$

So, $\boldsymbol{W}^{[1]}\boldsymbol{x}^{(i)} + \boldsymbol{b}^{[1]} = \boldsymbol{z}^{[1](i)}$, $\boldsymbol{Z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{X} + \boldsymbol{b}^{[1]}$.

4.6 Activation Functions

Given $\boldsymbol{x}$:

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$$\boldsymbol{a}^{[1]} = \sigma(\boldsymbol{z}^{[1]})$$

$$\boldsymbol{a}^{[1]} = g^{[1]}(\boldsymbol{z}^{[1]})$$

$$\boldsymbol{z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1]} + b^{[2]}$$

$$\boldsymbol{a}^{[2]} = \sigma(\boldsymbol{z}^{[2]})$$

$$\boldsymbol{a}^{[2]} = g^{[2]}(\boldsymbol{z}^{[2]})$$

We use $g(z)$ to denote activation function. Sometimes we can use other activation function like $tanh(z)$, ReLU ($max(0, z)$), etc., instead of $\sigma(z)$ function. Different layers of the neural network may have different activation function, so we can use $g^{[i]}(z)$ to denote the i-th layer's activation function of a neural network.

4.6.1 Pros and cons of activation functions

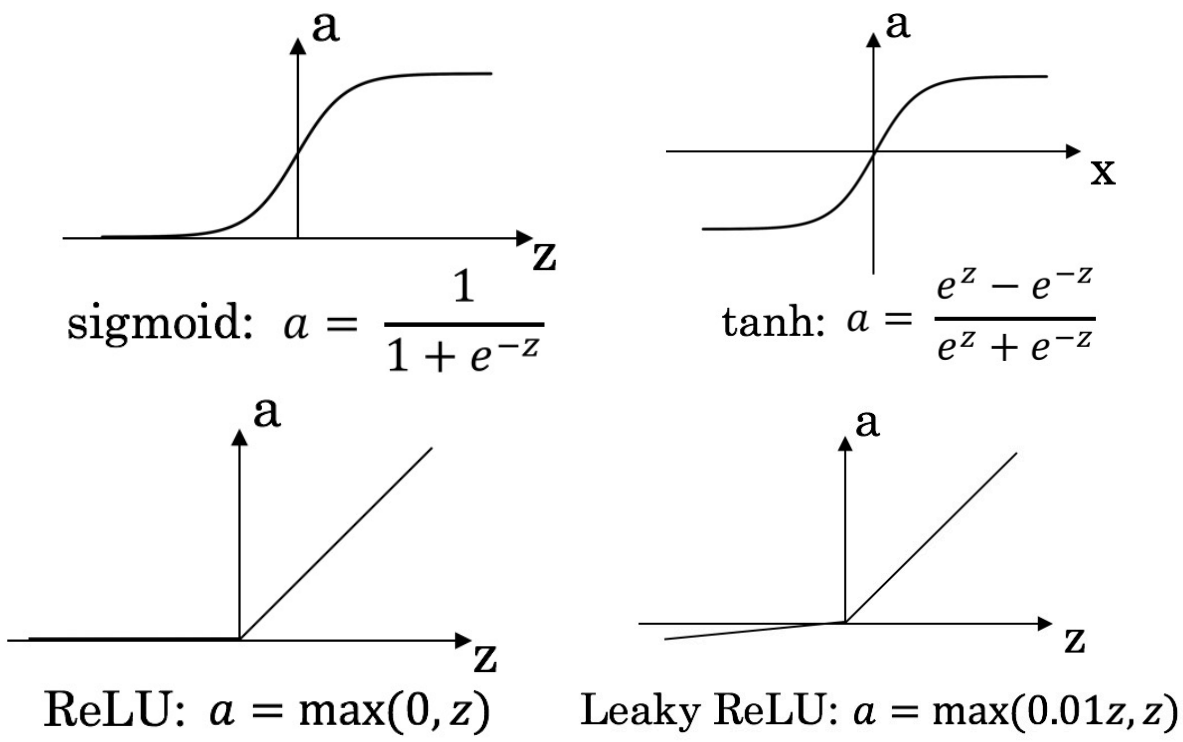


Figure 12: Activation functions

- One of the downside of both the sigmoid and tanh activation function is that if z is either very large or very small, then the gradient (derivative, slope) of the function becomes very small, this slows down the gradient descent.
- Never use sigmoid activation function, except for the output layer if you are doing binary classification. Tanh is pretty superior than sigmoid.
- If you don't have an idea to choose which activation function, choose ReLU as the default activation function, or you can try Leaky ReLU.

4.7 Why do you need non-linear activation functions?

If $g(z) = z$, sometimes we will call it "linear activation function" or "identity activation function". Given $\boldsymbol{x}$:

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$$\boldsymbol{a}^{[1]} = g^{[1]}(\boldsymbol{z}^{[1]}) = \boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$$\boldsymbol{z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1]} + b^{[2]} = \boldsymbol{W}^{[2]}(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}) + \boldsymbol{b}^{[2]} = \underbrace{(\boldsymbol{W}^{[2]}\boldsymbol{W}^{[1]})}_{\boldsymbol{W'}}\boldsymbol{x} + \underbrace{(\boldsymbol{W}^{[2]}\boldsymbol{b}^{[1]} + \boldsymbol{b}^{[2]})}_{\boldsymbol{b'}} = \boldsymbol{W'}\boldsymbol{x} + \boldsymbol{b'}$$

$$\boldsymbol{a}^{[2]} = g^{[2]}(\boldsymbol{z}^{[2]}) = \boldsymbol{z}^{[2]} = \boldsymbol{W'}\boldsymbol{x} + \boldsymbol{b'}$$

If you use linear activation function as the activation function of your neural network, then the neural network is just outputting a linear function of the input. It turns out that if you use a linear activation function or alternatively if you don't have an activation function, then no matter how many layers your neural network has always doing is just computing a linear activation function, so you might as well not have any hidden layers.

4.8 Derivatives of activation functions

4.8.1 Sigmoid activation function

$$g(z) = \frac{1}{1+e^{-z}} = a$$

$$g'(z) = \frac{\mathrm{d}}{\mathrm{d}z}g(z) = \frac{1}{1+e^{-z}}(1 - \frac{1}{1+e^{-z}}) = g(z)(1-g(z)) = a(1-a)$$

4.8.2 Tanh activation function

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = a$$

$$g'(z) = \frac{\mathrm{d}}{\mathrm{d}z}g(z) = 1 - (\tanh(z))^2 = 1 - a^2$$

4.9 ReLU

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 0, & \text{if } z < 0; \\ 1, & \text{if } z > 0; \\ \text{undefined}, & \text{if } z = 0. \end{cases} \xrightarrow{\text{implemented in sofware}} g'(z) = \begin{cases} 0, & \text{if } z < 0; \\ 1, & \text{if } z \geq 0. \end{cases}$$

4.10 Leaky ReLU

$$g(z) = \max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01, & \text{if } z < 0; \\ 1, & \text{if } z > 0; \\ \text{undefined}, & \text{if } z = 0. \end{cases} \xrightarrow{\text{implemented in sofware}} g'(z) = \begin{cases} 0.01, & \text{if } z < 0; \\ 1, & \text{if } z \geq 0. \end{cases}$$

4.11 Gradient descent for Neural Networks

Parameters: $n^{[0]} = n_x$, $n^{[1]}$, $n^{[2]} = 1$, $\underbrace{\boldsymbol{W}^{[1]}}_{(n^{[1]},n^{[0]})}$, $\underbrace{\boldsymbol{b}^{[1]}}_{(n^{[1]},1)}$, $\underbrace{\boldsymbol{W}^{[2]}}_{(n^{[2]},n^{[1]})}$, $\underbrace{\boldsymbol{b}^{[2]}}_{(n^{[2]},1)}$

Data: $\boldsymbol{X} = \underbrace{\begin{bmatrix} | & | & & | \\ \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \cdots & \boldsymbol{x}^{(m)} \\ | & | & & | \end{bmatrix}}_{(n^{[0]},m)}$, $\boldsymbol{Y} = \underbrace{\begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}}_{(1,m)}$

Cost Function:

$$J(\boldsymbol{W}^{[1]}, \boldsymbol{b}^{[1]}, \boldsymbol{W}^{[2]}, \boldsymbol{b}^{[2]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}, y)$$

repeat

`// Forward propagation`

$\boldsymbol{Z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{X} + \boldsymbol{b}^{[1]}$

$\boldsymbol{A}^{[1]} = g^{[1]}(\boldsymbol{Z}^{[1]})$

$\boldsymbol{Z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{A}^{[1]} + \boldsymbol{b}^{[2]}$

$\boldsymbol{A}^{[2]} = g^{[2]}(\boldsymbol{Z}^{[2]}) = \sigma(\boldsymbol{Z}^{[2]})$

`// Backpropagation`

$\mathrm{d}\boldsymbol{Z}^{[2]} = \boldsymbol{A}^{[2]} - \boldsymbol{Y}$

$\mathrm{d}\boldsymbol{W}^{[2]} = \frac{1}{m}\mathrm{d}\boldsymbol{Z}^{[2]}{\boldsymbol{A}^{[1]}}^T$

`// keepdims=True ensures that the shape of result is` $(n^{[2]}, 1)$`)`

$\mathrm{d}\boldsymbol{b}^{[2]} = \frac{1}{m}$np.sum($\mathrm{d}\boldsymbol{Z}^{[2]}$, axis=1, keepdims=True)

`// here` $*$ `means element-wise multiplication`

$\mathrm{d}\boldsymbol{Z}^{[1]} = {\boldsymbol{W}^{[2]}}^T\mathrm{d}\boldsymbol{Z}^{[2]} * {g^{[1]}}'(\boldsymbol{Z}^{[1]})$

$\mathrm{d}\boldsymbol{W}^{[1]} = \frac{1}{m}\mathrm{d}\boldsymbol{Z}^{[1]}{\boldsymbol{A}^{[0]}}^T = \frac{1}{m}\mathrm{d}\boldsymbol{Z}^{[1]}\boldsymbol{X}^T$

$\mathrm{d}\boldsymbol{b}^{[1]} = \frac{1}{m}$np.sum($\mathrm{d}\boldsymbol{Z}^{[1]}$, axis=1, keepdims=True)

until converge;

Algorithm 3: The gradient descent algorithm for neural networks

4.12 Backpropagation intuition (Optional)

According to Figure 6, we have got how to compute the gradient of logistic regression. By the chain rule, we can get:

$$\text{If } a = g(z), \text{then } \mathrm{d}z = \mathrm{d}a \cdot g'(z)$$

Logistic regression can be seen as a neural network with only one layer (exclude input layer), while the neural network in Figure 8 and Figure 9 contains two layers. To compute the derivatives for each single example, we can also use backpropagation and the chain rule.

At first, because $\mathcal{L}(\hat{y}, y) = -(y\log\hat{y} + (1-y)\log(1-\hat{y}))$, we can compute $\mathrm{d}a^{[2]}$:

$$\mathrm{d}a^{[2]} = \frac{\mathrm{d}\mathcal{L}(a^{[2]}, y)}{\mathrm{d}a^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

Then, $a^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$, according to the chain rule, we can compute $\mathrm{d}z^{[2]}$:

$$\mathrm{d}z^{[2]} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}a^{[2]}}\frac{\mathrm{d}a^{[2]}}{\mathrm{d}z^{[2]}} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}a^{[2]}}{g^{[2]}}'(z^{[2]}) = (-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}})[a^{[2]}(1-a^{[2]})] = a^{[2]} - y$$

Because $z^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1]} + b^{[2]}$, once we get $\mathrm{d}z^{[2]}$, we can continue to compute $\mathrm{d}\,\boldsymbol{W}^{[2]}$ and $\mathrm{d}b^{[2]}$:

$$\mathrm{d}\,\boldsymbol{W}^{[2]} = \mathrm{d}z^{[2]}\boldsymbol{a}^{[1]T}$$

$$\mathrm{d}b^{[2]} = \mathrm{d}z^{[2]}$$

To continue it, we can get $\mathrm{d}\boldsymbol{a}^{[1]} = \boldsymbol{W}^{[2]T}\mathrm{d}z^{[2]}$. In practice, the computation of $\mathrm{d}\boldsymbol{a}^{[1]}$ and $\mathrm{d}\boldsymbol{z}^{[1]}$ often collapses into one step. And we can check the dimensions, $\boldsymbol{W}^{[2]} \in \mathbb{R}^{(n^{[2]},n^{[1]})}$, $z^{[2]}, \mathrm{d}z^{[2]} \in \mathbb{R}^{(n^{[2]},1)}$, $\boldsymbol{z}^{[1]}, \mathrm{d}\boldsymbol{z}^{[1]} \in \mathbb{R}^{(n^{[1]},1)}$ (the $*$ here means element-wise multiplication):

$$\mathrm{d}\boldsymbol{z}^{[1]} = \mathrm{d}\boldsymbol{a}^{[1]} * g^{[1]\prime}(\boldsymbol{z}^{[1]}) = \underbrace{\underbrace{\boldsymbol{W}^{[2]T}}_{(n^{[1]},n^{[2]})}\underbrace{\mathrm{d}z^{[2]}}_{(n^{[2]},1)}}_{(n^{[1]},1)} * \underbrace{g^{[1]\prime}(\boldsymbol{z}^{[1]})}_{(n^{[1]},1)}$$

At last, we can get $\mathrm{d}\,\boldsymbol{W}^{[1]}$ and $\mathrm{d}\boldsymbol{b}^{[1]}$:

$$\mathrm{d}\,\boldsymbol{W}^{[1]} = \mathrm{d}\boldsymbol{z}^{[1]}\underbrace{\boldsymbol{x}^T}_{\boldsymbol{a}^{[0]T}}$$

$$\mathrm{d}\boldsymbol{b}^{[1]} = \mathrm{d}\boldsymbol{z}^{[1]}$$

4.12.1 Summary of gradient descent

For single example:

$$\mathrm{d}\boldsymbol{z}^{[2]} = \boldsymbol{a}^{[2]} - \boldsymbol{y}$$

$$\mathrm{d}\,\boldsymbol{W}^{[2]} = \mathrm{d}\boldsymbol{z}^{[2]}\boldsymbol{a}^{[1]T}$$

$$\mathrm{d}\boldsymbol{b}^{[2]} = \mathrm{d}\boldsymbol{z}^{[2]}$$

$$\mathrm{d}\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[2]T}\mathrm{d}\boldsymbol{z}^{[2]} * g^{[1]\prime}(\boldsymbol{z}^{[1]})$$

$$\mathrm{d}\,\boldsymbol{W}^{[1]} = \mathrm{d}\boldsymbol{z}^{[1]}\boldsymbol{x}^T$$

$$\mathrm{d}\boldsymbol{b}^{[1]} = \mathrm{d}\boldsymbol{z}^{[1]}$$

Vectorized implementation for m examples:

$$\mathrm{d}\boldsymbol{Z}^{[2]} = \boldsymbol{A}^{[2]} - \boldsymbol{Y}$$

$$\mathrm{d}\,\boldsymbol{W}^{[2]} = \frac{1}{m}\mathrm{d}\boldsymbol{Z}^{[2]}\boldsymbol{A}^{[1]T}$$

$$\mathrm{d}\boldsymbol{b}^{[2]} = \frac{1}{m}\text{np.sum}(\mathrm{d}\boldsymbol{Z}^{[2]}, \text{axis=1, keepdims=True})$$

$$\mathrm{d}\boldsymbol{Z}^{[1]} = \boldsymbol{W}^{[2]T}\mathrm{d}\boldsymbol{Z}^{[2]} * g^{[1]\prime}(\boldsymbol{Z}^{[1]})$$

$$\mathrm{d}\,\boldsymbol{W}^{[1]} = \frac{1}{m}\mathrm{d}\boldsymbol{Z}^{[1]}\boldsymbol{X}^T$$

$$\mathrm{d}\boldsymbol{b}^{[1]} = \frac{1}{m}\text{np.sum}(\boldsymbol{Z}^{[1]}, \text{axis=1, keepdims=True})$$

4.13 Random Initialization

When you are training a neural network, it's important to initialize your weights randomly. For logistic regression, it was okay to initialize the weights to 0, but for a neural network of intialized weights and parameters to all 0 and then apply gradient descent, it won't work.

As the Figure 13 shows, there is a shallow neural network with two input features, two hidden units and one output unit. Then we initialize all the weights and parameters to zero:

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad \boldsymbol{b}^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \boldsymbol{W}^{[2]} = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad \boldsymbol{b}^{[2]} = [0]$$

It turns out that initializing the bias terms $\boldsymbol{b}$ to all zero is exactly okay, but initializing $\boldsymbol{W}$ to all zero causes a problem. So the problem with this formalization is that for any example you give it, you'll have that $a_1^{[1]} = a_2^{[2]}$, and then when you compute backpropagation, it turns out that $\mathrm{d}z_1^{[1]} = \mathrm{d}z_2^{[2]}$.

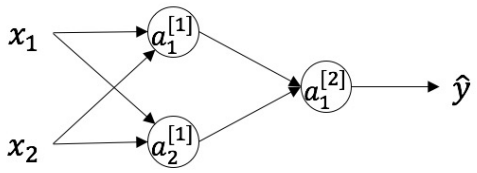


Figure 13: A shallow neural network with all weights and parameters initialized to 0

If we initialize the neural network this way, the hidden units $a_1^{[1]}$ and $a_2^{[1]}$ will be completely identical (symmetric). It turns out that after every single iteration of training, the two hidden units are still computing exactly the same function, both units have the same influence on the output unit. The $\mathrm{d}\boldsymbol{W}^{[1]}$ and $\boldsymbol{W}^{[1]}$ will looks like as the following:

$$\mathrm{d}\boldsymbol{W} = \begin{bmatrix} u & v \\ u & v \end{bmatrix} \qquad \boldsymbol{W}^{[1]} = \boldsymbol{W}^{[1]} - \alpha \mathrm{d}\boldsymbol{W} \qquad \boldsymbol{W}^{[1]} = \begin{bmatrix} r & s \\ r & s \end{bmatrix}$$

So in this case there is really no point to have more than one hidden unit, because all units are computing the same thing. However, that's not helpful, because you want different units compute different functions. The solution is to initialize the weights randomly:

$$\boldsymbol{W}^{[1]} = \text{np.random.randn(2, 2) * 0.01} \qquad \boldsymbol{b}^{[1]} = \text{np.zeros((2, 1))}$$

$$\boldsymbol{W}^{[2]} = \text{np.random.randn(1, 2) * 0.01} \qquad \boldsymbol{b}^{[2]} = \text{np.zeros((1, 1))}$$

Why we multiply a factor 0.01? That's because when we use the sigmoid or tanh activation function, if the weight is large, then the value input to the activation function will be large, so in that case the slope of the gradient is small and the gradient descent will be very slow.

5 Deep Neural Networks

5.1 Deep L-layer neural network

What is a deep neural network? As Figure 14 shows, logistic regresssion can be seen as one layer neural network. When we count the layers of the neural network, we don't count the input layer. Technically logistic regression is a quite shallow neural network, but over the last several years, the AI and machine learning community has realized that there are functions that very deep neural networks can learn while shallower are unable to.

Deep neural network notation Figure 15 shows a 4-layer neural network, with the input layer as "layer 0", three hidden layers "layer 1", "layer 2" and "layer 3", the output layer as "layer 4". Some notations should be remembered as the following:

L**:** # layers. In Figure 15, $L = 4$.

$n^{[l]}$**:** # units in layer l. In this case, $n^{[1]} = 5, n^{[2]} = 5, n^{[3]} = 3, n^{[4]} = n^{[L]} = 1, n^{[0]} = n_x = 3$.

$\boldsymbol{a}^{[l]}$**:** activations in layer l. In propagation, we will implement $\boldsymbol{a}^{[l]} = g^{[l]}(\boldsymbol{z}^{[l]})$. For single example, $\boldsymbol{a}^{[0]} = \boldsymbol{x}$, $\boldsymbol{a}^{[L]} = \hat{\boldsymbol{y}}$

$\boldsymbol{W}^{[l]}$**:** weights for $\boldsymbol{z}^{[l]}$.

5.2 Forward Propagation in a Deep Network

We still look at the Figure 15, see the forward propagation in this neural network.

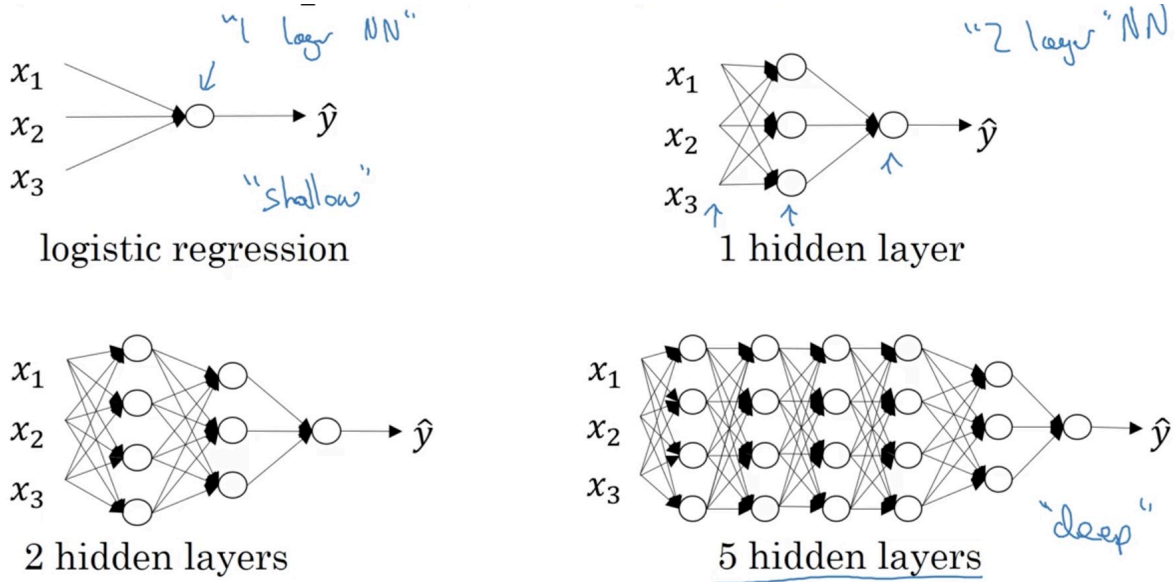


Figure 14: Examples of neural networks

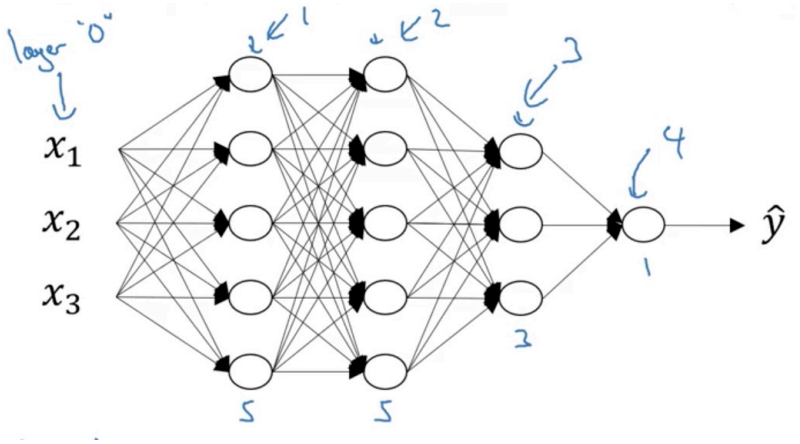


Figure 15: A 4-layer neural network

For every single example,

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{a}^{[0]} + \boldsymbol{b}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$$\boldsymbol{a}^{[1]} = g^{[1]}(\boldsymbol{z}^{[1]})$$

$$\boldsymbol{z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1]} + \boldsymbol{b}^{[2]}$$

$$\boldsymbol{a}^{[2]} = g^{[2]}(\boldsymbol{z}^{[2]})$$

$$\boldsymbol{z}^{[3]} = \boldsymbol{W}^{[3]}\boldsymbol{a}^{[2]} + \boldsymbol{b}^{[3]}$$

$$\boldsymbol{a}^{[3]} = g^{[3]}(\boldsymbol{z}^{[3]})$$

$$\boldsymbol{z}^{[4]} = \boldsymbol{W}^{[4]}\boldsymbol{a}^{[3]} + \boldsymbol{b}^{[4]}$$

$$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$$

Then we can get the general rule for forward propagation:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

For the whole training set, in a vectorized way:

$$Z^{[1]} = W^{[1]} A^{[0]} + b^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$
$$Z^{[3]} = W^{[3]} A^{[2]} + b^{[3]}$$
$$A^{[3]} = g^{[3]}(Z^{[3]})$$
$$Z^{[4]} = W^{[4]} A^{[3]} + b^{[4]}$$
$$A^{[4]} = g^{[4]}(Z^{[4]}) = \hat{Y}$$

We can put the left into an explicit for-loop:

for $l = 1$ **to** 4 **do**

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

end

5.3 Getting your matrix dimensions right

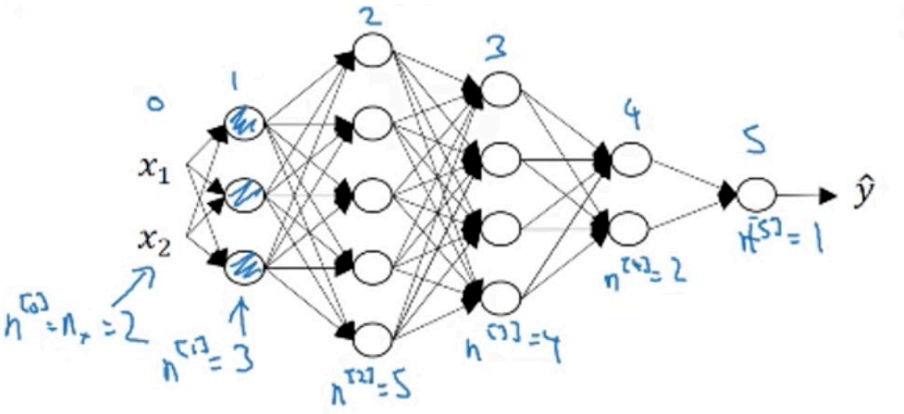


Figure 16: A 5-layer neural network

In Figure 16, there is a five-layer neural network, with $n^{[0]} = n_x = 2$, $n^{[1]} = 3$, $n^{[2]} = 5$, $n^{[3]} = 4$, $n^{[4]} = 2$ and $n^{[5]} = 1$. The checking of the matrix dimensions can be seen as following.

For every single example

$$\underbrace{z^{[1]}}_{(n^{[1]},1)=(3,1)} = \underbrace{\underbrace{W^{[1]}}_{(n^{[1]},n^{[0]})=(3,2)} \underbrace{x}_{(n^{[0]},1)=(2,1)}}_{(3,1)} + \underbrace{b^{[1]}}_{(n^{[1]},1)=(3,1)}$$

$$\underbrace{\boldsymbol{Z}^{[2]}}_{(n^{[2]},1)=(5,1)} = \underbrace{\boldsymbol{W}^{[2]}}_{(n^{[2]},n^{[1]})=(5,3)} \underbrace{\boldsymbol{a}^{[1]}}_{(3,1)} + \underbrace{\boldsymbol{b}^{[2]}}_{(n^{[2]},1)=(5,1)}$$

.....

$$\boldsymbol{W}^{[3]} : (n^{[3]}, n^{[2]}) = (4, 5) \qquad \boldsymbol{W}^{[4]} : (n^{[4]}, n^{[3]}) = (2, 4) \qquad \boldsymbol{W}^{[5]} : (n^{[5]}, n^{[4]}) = (1, 2)$$

$$\boldsymbol{z}^{[3]}, \boldsymbol{a}^{[3]}, \boldsymbol{b}^{[3]} : (n^{[3]}, 1) = (4, 1) \qquad \boldsymbol{z}^{[4]}, \boldsymbol{a}^{[4]}, \boldsymbol{b}^{[4]} : (n^{[4]}, 1) = (3, 1) \qquad \boldsymbol{z}^{[5]}, \boldsymbol{a}^{[5]}, \boldsymbol{b}^{[5]} : (n^{[5]}, 1) = (1, 1)$$

The general formula to check is that when you're implementing the matrix $\boldsymbol{W}$ for layer l, the dimension of that matrix will be $(n^{[l]}, n^{[l-1]})$. And if you're implementing the back-propagation, the matrix $\mathrm{d}\boldsymbol{W}$ will have the same dimension with $\boldsymbol{W}$, the vector $\mathrm{d}\boldsymbol{b}$ will have the same dimension with $\boldsymbol{b}$. So for each single example, the formula is:

$$\boldsymbol{W}^{[l]} : (n^{[l]}, n^{[l-1]}) \qquad \mathrm{d}\boldsymbol{W}^{[l]} : (n^{[l]}, n^{[l-1]})$$

$$\boldsymbol{z}^{[l]}, \boldsymbol{a}^{[l]}, \boldsymbol{b}^{[l]} : (n^{[l]}, 1) \qquad \mathrm{d}\boldsymbol{z}^{[l]}, \mathrm{d}\boldsymbol{a}^{[l]}, \mathrm{d}\boldsymbol{b}^{[l]} : (n^{[l]}, 1)$$

Vectorized implementation

$$\underbrace{\boldsymbol{Z}^{[1]}}_{(n^{[1]},m)} = \underbrace{\underbrace{\boldsymbol{W}^{[1]}}_{(n^{[1]},n^{[0]})} \underbrace{\boldsymbol{X}}_{(n^{[0]},m)}}_{(n^{[1]},m)} + \underbrace{\boldsymbol{b}^{[1]}}_{(n^{[1]},1)\to(n^{[1]},m)}$$

.....

Stack all m training examples together, then we can get the vectorized version of the formula:

$$\boldsymbol{W}^{[l]} : (n^{[l]}, n^{[l-1]}) \qquad \mathrm{d}\boldsymbol{W}^{[l]} : (n^{[l]}, n^{[l-1]})$$

$$\boldsymbol{b}^{[l]} : (n^{[l]}, 1) \qquad \mathrm{d}\boldsymbol{b}^{[l]} : (n^{[l]}, 1)$$

$$\boldsymbol{Z}^{[l]}, \boldsymbol{A}^{[l]} : (n^{[l]}, m) \qquad \mathrm{d}\boldsymbol{Z}^{[l]}, \mathrm{d}\boldsymbol{A}^{[l]} : (n^{[l]}, m)$$

5.4 Why deep representations?

5.4.1 Intuition about deep representation

Like Figure 17 shows, if you are building a system for face recognition or face detectioin, here's what a deep neural network could be doing. Perhaps you input a picture of a face then the first layer of the neural network you can think of as maybe being a feature detector or an edge detector.

In this example, there are about 20 units in the hidden layer. So the 20 hidden units are visualized by these little square boxes. So for example, this little visualization in the first hidden layer represents the hidden unit is trying to figure out where the edges of that orientation in the image.

Now, let's think about where the edges in this picture by grouping together pixels to form edges. It can then detect the edges and group edges together to form parts of the faces. So for example, you might have a neuron trying to see if it's finding an eye, or a different neuron trying to find that part of the nose. So by putting together lots of edges, it can start to detect different parts of faces.

Finally, by putting together different parts of faces, like an eye or a nose or an ear or a chin, it can try to recognize or detect different types of faces.

So intuitively, you can think of the earlier layers of the neural network as detecting simple functions, like edges. And then composing them together in the later layers of a neural network so that it can learn more and more complex functions. These visualizations will make more sense when we talk about convolutional nets.

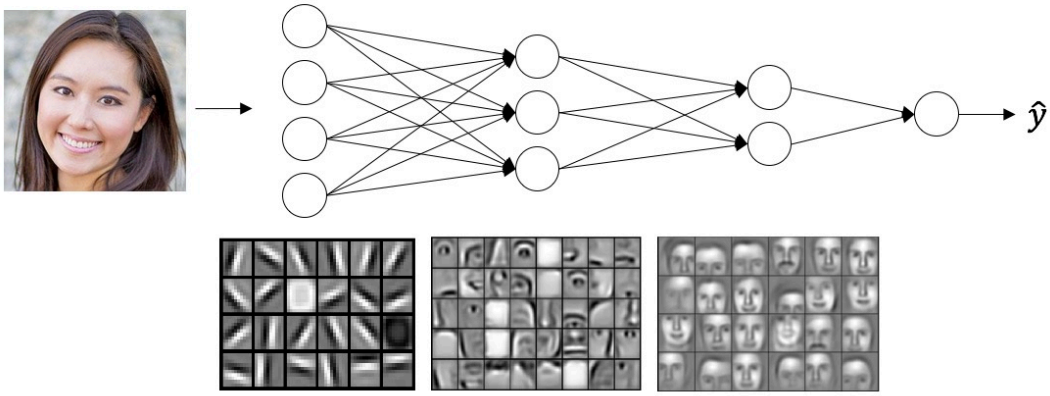


Figure 17: Intuition about deep representation

5.4.2 Circuit theory and deep learning

Informally: There are functions you can compute with a "small" (hidden units) L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

5.5 Building blocks of deep neural networks

For the layer l in the deep neural network:

Parameters: $\boldsymbol{W}^{[l]}$, $\boldsymbol{b}^{[l]}$.

Forward: input: $\boldsymbol{a}^{[l-1]}$; output: $\boldsymbol{a}^{[l]}$; cache ($\boldsymbol{z}^{[l]}$).
$\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}$
$\boldsymbol{a}^{[l]} = g^{[l]}(\boldsymbol{z}^{[l]})$
Vectorized version:
$\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{A}^{[l-1]} + \boldsymbol{b}^{[l]}$
$\boldsymbol{A}^{[l]} = g^{[l]}(\boldsymbol{Z}^{[l]})$

Backward: input: $\mathrm{d}\boldsymbol{a}^{[l]}$ (get from cache ($\boldsymbol{z}^{[l]}$)); output: $\mathrm{d}\boldsymbol{a}^{[l-1]}$, $\mathrm{d}\,\boldsymbol{W}^{[l]}$, $\mathrm{d}\boldsymbol{b}^{[l]}$;
$\mathrm{d}\boldsymbol{z}^{[l]} = \mathrm{d}\boldsymbol{a}^{[l]} * g^{[l]\prime}(\boldsymbol{z}^{[l]})$
$\mathrm{d}\,\boldsymbol{W}^{[l]} = \mathrm{d}\boldsymbol{z}^{[l]}\boldsymbol{a}^{[l-1]}$
$\mathrm{d}\boldsymbol{b}^{[l]} = \mathrm{d}\boldsymbol{z}^{[l]}$
$\mathrm{d}\boldsymbol{a}^{[l-1]} = \boldsymbol{W}^{[l]T}\mathrm{d}\boldsymbol{z}^{[l]}$
Vectorized version:
$\mathrm{d}\boldsymbol{Z}^{[l]} = \mathrm{d}\boldsymbol{A}^{[l]} * g^{[l]\prime}(\boldsymbol{Z}^{[l]})$
$\mathrm{d}\,\boldsymbol{W}^{[l]} = \frac{1}{m}\mathrm{d}\boldsymbol{Z}^{[l]}\boldsymbol{A}^{[l-1]T}$
$\mathrm{d}\boldsymbol{b}^{[l]} = \frac{1}{m}$np.sum($\mathrm{d}\boldsymbol{Z}^{[l]}$, axis=1, keepdims=True)
$\mathrm{d}\boldsymbol{A}^{[l-1]} = \boldsymbol{W}^{[l]T}\mathrm{d}\boldsymbol{Z}^{[l]}$

The process above can be concluded as a block in Figure 19.

5.6 Parameters vs Hyperparameters

5.6.1 What are hyperparameters?

Paramters: $\boldsymbol{W}^{[1]}$, $\boldsymbol{b}^{[1]}$, $\boldsymbol{W}^{[2]}$, $\boldsymbol{b}^{[2]}$, $\boldsymbol{W}^{[3]}$, $\boldsymbol{b}^{[3]}$ ...

Hyperparameters: learning rate α; # iterations; # hidden layer L; # hidden units $n^{[1]}$, $n^{[2]}$, ...; choice of activation function.

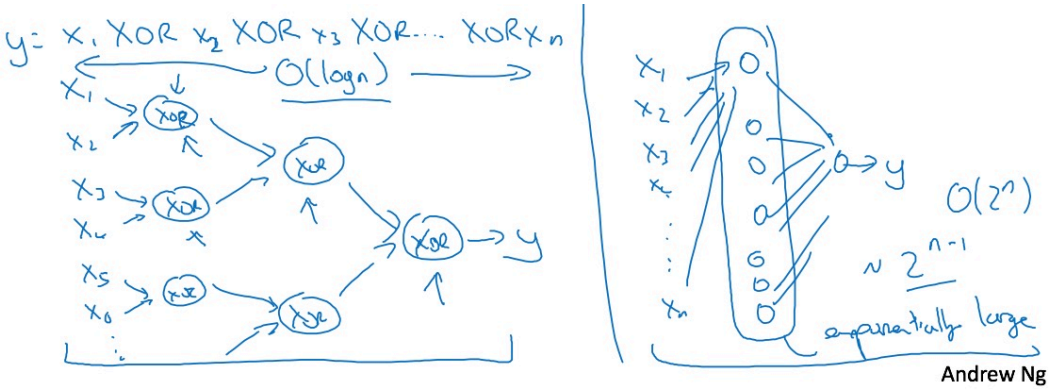


Figure 18: Use deep neural networks and shallower neural networks to implement the XOR of n input. For deep neural network (the left one), to compute the XOR, the depth of the network will be on the order of $O(\log n)$, so the number of nodes, or the number of circuit components or the number of gates in this neural network is not that large. But if you are forced to compute the function with just one hidden layer, then in order to get the parity of XOR to compute this XOR function, the size of this hidden layer will need to be exponentially large (2^{n-1}), because essentially you need to exhaustively enumerate all 2 to the n possible configurations.

Hyperparamters are parameters that control parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. In later course, we will learn other hyperparameters such as momentum, mini-batch size and regularization parameters.

5.6.2 Applied deep learning is a very empirical process

Applied deep learning today is a very empirical process. For example, in Figure 20 you have an idea that set $\alpha = 0.01$, then you implement it, try out and see how that works, and then based on the outcome you might say you know that I have changed my mind, then you do the process again to try $\alpha = 0.05$. You might plot the cost J of the number of iterations to see which is the best learning rate.

The 'empirical process' in the title means that you have to try a lit if things and see what works. When starting on a new problem, just try out a range of values and see what works. In the next course, we'll see some systematic ways for trying out a range of values of hyperprameters.

Secondly, if you're working on one application for a long time, maybe you're working on online advertising as you make progress on the problem is quite possible there the best value for the learning rate, a number of hidden units and so on might change, even if you tune your system to the best value of hyperparameters today, it's possible that you find taht the best value might change year from now.

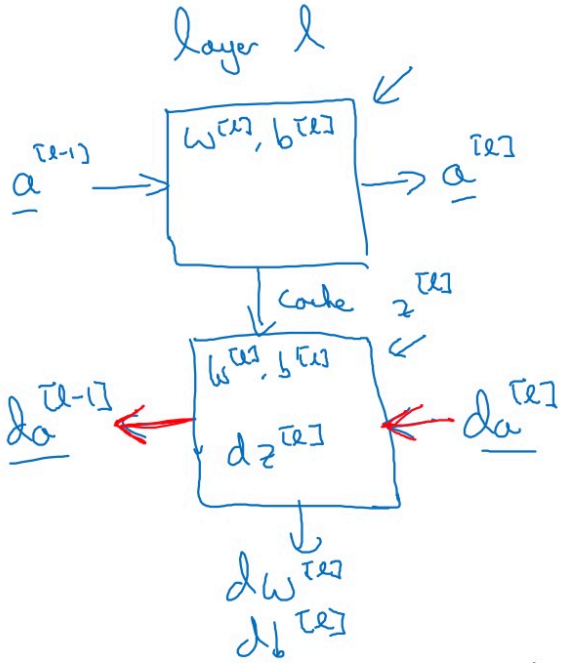


Figure 19: A block of forward and backward process in a deep Neural network

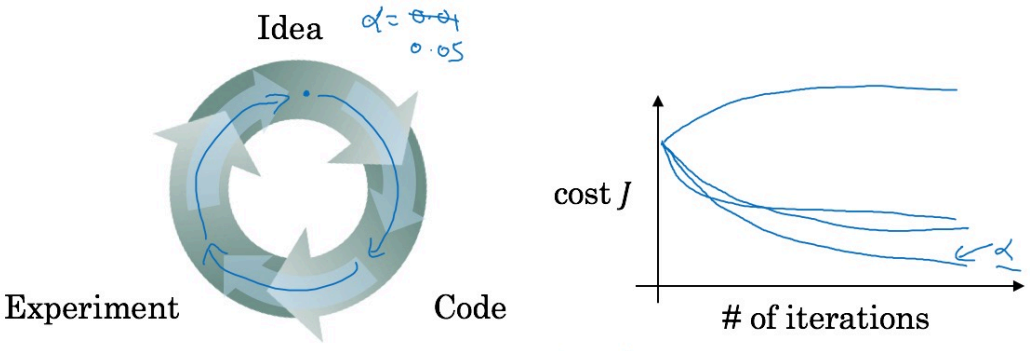


Figure 20: The process of try different hyperparamters in a deep Neural network